Handbook for Authors of Transition Statements

by

Dick Stevens

July 28, 2002

I Introduction

The purpose of this handbook is to describe for the PGMT (Processing Graph Method Tool) GUI (Graph User Interface) operator how to write a transition statement.

The style of this handbook is intended to be casual; we write as if we were speaking directly to the operator. Accordingly, the term *you* refers to the operator.

As described in the GUI Handbook, the GUI provides a means whereby you, the operator, can prescribe a PGM processing graph. The GUI stores the information in your processing graph in a *GSF* (Graph State File). The Translator generates C++ source code for your processing graph. When this code is compiled and linked with the PGMT source code, along with your hand-written command program, it may be executed to process data as prescribed by your processing graph.

Some diagnostics are available within the GUI to help identify certain errors in your processing graph. These diagnostics are necessarily incomplete; some errors can only be caught later – at compile time or at run time.

To provide some context, we will discuss what Translator does automatically, followed by a description of what you, the operator, must do. We will then give a list of the functions provided by PGMT that may be called within the transition statement. Any additional functions, such as might be found in a third-party library or ones you may write yourself, may also be used, and we will explain how to do that.

We give an example of code that will construct a token with height 3 in Appendix A. Appendix B shows code that can be used to access the same token.

II What you Should Know

We assume that you are familiar with the *PGM Spec* (Processing Graph Method 2.1 Semantics, by David J. Kaplan and Richard S. Stevens, July 18, 2002). We assume you are also familiar with the *PGMT User's Manual* (Processing Graph Method Tool (PGMT) User's Manual, by Wendell Anderson, March 20, 2002) as well as the *GUI Handbook* (Processing Graph Method - GUI Handbook, by Michal Iglewski, July 17, 2002).

We assume that you have some programming experience with C++. In particular, it would be helpful for you to know about the class templates *vector* and *deque* as defined in the *STL* (Standard Template Library). There are numerous books about the STL available. We will not cite any here.

In the management of a family in PGMT, we usually use two data structures. The first is the *data array*. This is stored in a vector, as defined in the STL. The data are the leaves, stored in row-major order (i.e., last index varying most rapidly). We store the family tree in a separate data structure called the *Descriptor*. In PGMT there is a class called GCL_Descriptor, which defines methods for accessing the family tree, which can be used, for example to calculate the offset of any leaf from its family indices. We do not use the standard C++ notation for indexing in the code. Rather we use a vector of indices, as we will describe below.

III What the Translator Does for You

From your GSF, the Translator generates a C++ header file containing a set of class definitions. There is a class definition for your processing graph, generated from its Exterior Form, and a class definition for each Transition Exterior Form. This header file contains all of the header information as well as all the specific method bodies for each class. The reason for putting all of the method bodies in the header file is because of the possibility that some of the class definitions may result in class templates. Different C++ compilers handle class templates in different ways, creating a number of problems with the traditional approach of defining the classes and prototypes in header files and the method bodies in files separate form the header files. Our choice of putting everything into a header file follows the lead of the Standard Template Library.

The class definition for each of your transition exteriors includes a method called *tranStmt*() (short for Transition Statement). The Translator generates the body of this method, including some declarations of variables as well as the code that you write in the Transition Statement Form in the GUI. It is important to understand that the Transition Statement executes exactly once during each execution of the Transition.

The automatically generated variable declarations are the names of the input ports and output ports that you defined in the Transition Exterior Form. Exactly what each of these port names represents depends on the port definition, as described below:

For an input port or output port

- If the port family has height 0 (i.e., the input port has no family tree):
  - If the mode of the port has height 0 (i.e., the tokens have height 0), the port name is declared to be of the base type.

    - If the port is an input port, you may assume that the variable has been assigned the value of the token read in for the current execution of the Transition.
    - If the port is an output port, you must write the Transition Statement to assign the output token value to this variable.
  - If the mode of the port has height > 0 (i.e., the tokens have height > 0), the port name is declared to be the workspace associated with the port. The *workspace* is variable defined by a class in PGMT for the purpose of managing tokens with height > 0. We describe the workspace methods for managing tokens below under the heading *Functions Available To You*.
    - If the port is an input port, you may assume that the workspace contains the token read in for the current execution.
    - If the port is an output port, you must write the Transition Statement to create the output token within this workspace.
- If the port family has height > 0 (i.e., the input port has a non-empty family tree), then there is a family of possibly many ports. For each such port family, the port family name represents a single workspace, which contains a single token that is the aggregate assembly of all the tokens. The height of the token contained in this workspace is the sum of the mode height (i.e., the height of each token read at each port in the port family) and the port family height. Moreover, the appropriate number of upper levels of this token's family tree must match the port family tree.
  - If the port is an input port family, then you may assume that the workspace contains the token assembled from all the tokens read at the input ports in the port family. The upper levels of the token's family tree will match the port family tree.
  - If the port is an output port family, then the token you construct will be automatically disassembled into separate tokens, each to be produced at the respective output port in the port family. A run-time error will occur if the upper levels of the token's family tree do not match the port family tree.

IV What You Must Do

As noted in the previous section, you must write the body of the Transition Statement to define the output token for each of the Transition's output ports. If the output token's height is zero, this is simply a matter of assigning the value to the respective variable whose name is that of the output port. If the output token's height is non-zero (i.e., the port mode or the port family has non-zero height), then you must use the workspace (variable name is that of the output port) to construct the token to be output. If the port family has non-zero height, you must also be sure that the upper levels of the tokens' family tree match the output port's family tree.

Also, you must ensure that you do not create any memory leaks. While PGMT was written with great care to delete all memory allocated within the PGMT code, it is impossible for PGMT to delete memory allocated within the code that you write in the Transition Statement. Unfortunately this is a consequence of our choice of C++ as the language for PGMT. There is no garbage collection within the standard C++ language. If you allocate memory and fail to delete it, it will leak. If your Transition Statement leaks memory, no matter how slowly, and if your graph runs long enough, it will eventually use up all available memory.

One final caution about memory leaks: It is as bad to delete memory that has already been deleted as it is not to delete memory that has been allocated. If you delete memory that has already been deleted, then it is possible for that deleted memory to be reallocated and your deleting it will cause it to be returned to the memory heap while you intend to continue using it.

To assist in finding and eliminating memory leaks, there are software tools available. We do not recommend any specific ones here.

V Functions Available to You

In this section, we speak of an input token or output token as seen from within the transition statement. Thus, an input or output token will have height 0 if and only if both the port mode and the port family have height 0.

For each input port and each output port where the token has height 0, the port name simply represents the value of the respective token read from or produced to the port. No additional PGMT functions are needed to support management of input or output tokens with height 0.

The functions we describe below are all methods of classes. Most of the classes are templates with base type T (base type being the base type of the token). For each method we will give the class of which it is a method as well as the method's prototype and a brief description of the function's arguments and its returned value.

The class templates *vector* and *deque* are defined in the Standard Template Library, which is bundled with the GNU C++ compiler.

For each input port and each output port where the token has non-zero height, the port name represents the respective workspace (i.e., the workspace itself, not a pointer to it). First we list the PGMT

functions that may be used to uncover the structure of an input token. Following that, we list the PGMT functions that may be used to construct an output token.

V.A Functions for Uncovering the Structure of an Input Token

In this section we give the prototypes of various functions that you can use in the transition statement.

V.A.1 Methods of the class WorkSpace_T<T>:

For prototypes and definitions, refer to files:

GCL_WorkSpace_T.h,
GCL_WorkSpace.h,
GCL_WorkSpace.cpp.

```
const GCL_Token_T< T > *getToken () const;
```

Returns a pointer to the token stored in the workspace.

```
const GCL_Descriptor *getDescriptor () const;
```

Returns a pointer to the token's descriptor.

```
unsigned int getNumChildren () const;
```

Returns the number of Children in the token.

```
unsigned int getNumLeaves () const;
```

Returns the number of Leaves in the token.

```
unsigned int getDepth () const;
```

Returns the token height (attribute of the token).

```
unsigned int get__tokenHt () const;
```

Returns the token height (attribute of the WorkSpace).

```
const GCL_BaseType  get__tokenBaseType () const;
```

returns the token base type (attribute of the WorkSpace - really the MPI base type).

```
deque<const GCL_Token_T<T> *> * unpack ();
```

returns an allocated deque of allocated token pointers to the children of the token.

V.A.2 Methods of the class Token_T<T>:

For prototypes and definitions, refer to files:

GCL_Token_T.h,
GCL_Token.h,
GCL_Token.cpp.

```
const GCL_Descriptor *get__descriptor () const;
```
returns a pointer to the descriptor of the token.

```
unsigned int getLeafCount () const;
```
returns the total number of leaves in the token.

```
unsigned int getChildCount () const;
```
returns the number of children (i.e., the number of rows).

```
unsigned int getDepth () const;
```
returns the height of the token.

```
int getLowerBound () const;
```
returns the lower bound of the first index.

```
T *getLeafArray ();
```
returns a pointer to the leaves, (A one dimensional array. We use *offset* to refer to the index into this array. The offset may be any integer value between 0 and leafCount – 1.) The pointer returned by this method may be used to pass the token data to a C routine as an array. You should be cautioned against passing this array directly if the routine will modify the data in place. It will cause the token's data array to be altered.

```
T getLeaf (const vector<int> & indices) const;
```
returns the leaf whose indices are in the vector argument.

Additional functions not of any class may be used to assemble and disassemble tokens. See section V.C below for descriptions of these functions.

V.A.3 Methods of the class Descriptor:

The methods of this class may be useful if you want to find the details of a family tree directly from the descriptor. In particular, the lower and upper index bounds can be found for the family descriptor. But the lower and upper bounds of its children are not directly accessible from the descriptor. However, you

can use the "makeChildren" method below to obtain descriptors of the children. Then from each such descriptor, you can find out the lower and upper bounds.

Another approach would be to call the function "getIndices" below with the "offset" successively set the to 0, 1, …, leafCount – 1, where the leafCount is the total number of leaves in the family. By examining the index vectors that are returned, you can infer the lower and upper bounds of the children and all the descendants.

For prototypes and definitions, refer to files:

GCL_Descriptor.h,
GCL_Descriptor.cpp.

```
int getLowerBound() const;
```

returns the lower bound

```
unsigned int getChildCount() const;
```

returns the number of children

```
unsigned int getLeafCount();
```

returns the number of leaves

```
unsigned int getDepth();
```

returns the height

```
deque<const GCL_Descriptor *> * makeChildren() const;
```

returns a deque of descriptors of the children. This function constructs a deque of pointers to descriptors and returns a pointer to it. You will have to call the destructor for each child descriptor and for the deque itself to avoid a memory leak. Destroying the child descriptors is facilitated by the following function "clearDeque" described below in V.C.

```
unsigned int getOffset(const vector <int> & indices) const;
```

returns the data offset corresponding to given index vector.

```
vector<int> * getIndices(unsigned int offset) const;
```

returns the index vector corresponding to given leaf offset. The index vector returned is allocated by the method, and you are responsible for deleting it when you are through with it.

V.B Functions for Constructing an Output Token

V.B.1 Methods of the class WorkSpace_T<T>:

For prototypes and definitions, refer to files:

GCL_WorkSpace_T.h,
GCL_WorkSpace.h,
GCL_WorkSpace.cpp.

```
bool checkToken () const;
```

Returns true if the WorkSpace contains a token; otherwise returns false.

```
void storeToken (const GCL_Token_T< T > *);
```

Stores the token into the WorkSpace.

```
void pack (deque< const GCL_Token_T< T > * > &,
           unsigned int inDepth);
```

Assembles the tokens whose pointers are in the deque and stores the resulting token in the WorkSpace. The second argument inDepth must match the common depth of the tokens in the deque. This argument is necessary if the deque is empty. The token height will be inDepth + 1.

```
const GCL_Descriptor * getPortFamilyDescriptor() const;
```

Returns a pointer to the descriptor of the port family. This is useful if you have to construct a token for an output port family. This descriptor will be the parent of the assembled output token (see the function "assemble" in V.C below.

V.B.2 Methods of the class Token_T<T>:

For prototypes and definitions, refer to files:

GCL_Token_T.h,
GCL_Token.h,
GCL_Token.cpp.

```
GCL_Token_T (const GCL_Token_T<T> &);
```

Constructs a token that is a copy of the given token.

```
GCL_Token_T (const GCL_Descriptor *);
```

Constructor that makes a token with the given descriptor. The data array is filled with the default value of the base type (usually 0).

```
GCL_Token_T ( const deque <const GCL_Token_T<T> *> & inDeque,
              const GCL_Descriptor & descriptor,
```

```
        unsigned int tokenDepth );
```

Constructs a token that is assembled from the given descriptor (second argument) for the upper levels and the deque of tokens (first argument) for the lower levels. The tokenDepth (third argument) must match common height of the tokens in the deque. This argument is necessary to cover the case where the deque is empty. The number of leaves in the descriptor must match the number of tokens in the deque.

```
unsigned int setValues (T *array, unsigned int count = 1,
      unsigned int offset = 0);
```

Sets the values consecutively in the array into the data array of the token. The number of values stored is given by the second argument, and the first element of the array is stored at the offset position. Memory outside the data array of the token is protected; i.e., the function copies the values until the argument count is reached or until the last element of the data array is filled, whichever occurs first. The value returned is the actual number of values stored.

```
T *getLeafArray ();
```

returns a pointer to the leaves, (A one dimensional array. We use *offset* to refer to the index into this array. The offset may be any integer value between 0 and leafCount – 1.)

```
void putLeaf(const vector<int> & indices, T value);
```

assigns the value to the leaf in the token identified by the index vector "indices".

Additional functions not of any class may be used to assemble and disassemble tokens. See section V.C below for descriptions of these functions.

V.B.3 Methods of the class Descriptor:

The methods of this class may be useful if you want to construct a descriptor for a token directly from the descriptor. You may then use this descriptor as an argument to a constructor function above and then fill the leaves into the data array.

For prototypes and definitions, refer to files:

GCL_Descriptor.h,
GCL_Descriptor.cpp.

First we give a list of some of the useful descriptor constructors:

```
GCL_Descriptor ();
```

This is the default constructor: It constructs a family with depth 0; i.e., descriptor of a scalar.

```
GCL_Descriptor (const GCL_Descriptor & descriptor);
```
This is the copy constructor: It makes a descriptor that is a copy of the argument.

```
GCL_Descriptor (const GCL_Descriptor & descriptor,
            const_vector_int & indices);
```
Constructor of a descendant of a given descriptor. The vector of indices identifies the descendant.

```
GCL_Descriptor (unsigned int inDepth);
```
Constructor: empty family with given depth.

```
GCL_Descriptor (int lowerBound, unsigned int childCount);
```
Constructor with lower bound and child count, The height will be 1.

```
GCL_Descriptor (const vector<unsigned int> & dimensions,
            int lowerBound = 0);
```
Constructor of completely regular descriptor. In a *regular* descriptor, all children have identical descriptors. In a *completely regular* descriptor, all descendants at each level have identical descriptors. In simple terms, a completely regular descriptor is rectangular in every dimension.

```
GCL_Descriptor (const deque<const GCL_Descriptor *> & indeque,
            unsigned int inDepth, int lowerBound = 0);
```
Constructor with given deque of children.

```
const GCL_Descriptor * getPortFamilyDescriptor() const;
```
Returns a pointer to the associated Port Family Descriptor.

V.C Useful functions not of any class

```
template <class T> void clearDeque (deque<T *> & myDeque);
```
Clears a deque of pointers to things of any type. Specifically, deletes all of the items to which the deque points and empties the deque. If the deque, itself, was allocated and you are finished with it, then you should delete the deque after clearing it.

```
const GCL_Descriptor * assemble (const GCL_Descriptor & parent,
          const deque <const GCL_Descriptor *> & children,
          unsigned int childDepth);
```
Assembles a new descriptor from the parent to make the upper levels and attaching the respective child descriptors in the deque to the leaves of the parent. The third argument gives the common height of the children. This argument is necessary to account for the possibility that the

parent has no leaves – in which case the deque of children must be empty. For this function to be successful, the deque must have the same number of children as there are leaves in the parent.

```
const GCL_Descriptor * disassemble (
          const GCL_Descriptor & descriptor,
          deque <const GCL_Descriptor *> & children,
          unsigned int childDepth);
```

Disassembles the given descriptor so that the children have the depth given in the third argument. The function returns a pointer to an allocated descriptor for the upper levels, and pointers to the allocated child descriptors are placed in the deque in the second argument.

```
GCL_Token_T<T> * pack (const deque<const GCL_Token_T<T> *> & inDeque,
                                  unsigned int inDepth);
```

Takes a deque of tokens and packs them into a single token, returning a pointer to the resulting token. The second argument inDepth must match the common height of the tokens in the deque.

```
deque<const GCL_Token_T<T> *> * unpack (const GCL_Token_T<T> &
inToken);
```

Takes a token with height > 0 and unpacks it into a deque of token pointers with height one lower than the given token. Returns a pointer to the resulting deque.

VI How To Add Additional Functions and Data Types

You may want to define some additional functions of your own for use in transition statements. Or you may want to use one or more C or C++ routines that you obtain from a third-party library. Finally, you may even want to define your own data types. Once the data types are defined, you will want to incorporate the functions and data types into your PGMT application. You must do the following:

- Enter the header files containing the prototypes of the class and data type definitions in the Type List Form (See the GUI Handbook).
- Make sure that the body files (*.c or *.cpp) are compiled and linked with your application.
- Make sure that any data types that you define in separate files (classes and structures) are processed by MTOOL in order to make sure that the appropriate MPI data types are constructed. The resulting files generated by MTOOL must also be compiled and linked with your application. See the document "MTOOL/PGMT Integration, by Roger Hillson".

VII Command Program Interface with the graph

All of the functions described in section V above are available for use in the Command Program. Each Graph Input Port and Output Port has an associated Work Space that is accessible to the Command Program, just as the Transition Input Ports and output ports are accessible to the Transition Statement.

There are some difference that bear mention.

- In the mode of a graph port, if the token height is 0, then the Command Program Writer must enter the token's leaf value into the Work Space for a Graph Input Port, and he/she must retrieve the token's leaf value from the Work Space for a Graph Output Port. This differs from the Transition Statement in that for a token height of 0, the Transition Port Name represents the leaf value directly.
-

Appendix A
Examples: Access of Tokens

In a Transition Statement, one will usually access a token stored in a Work Space associated with a Transition Input Port. In a Command Program, the Work Space is associated with a Graph Output Port. The following examples assume that you are accessing a token that is contained in a Work Space.

There are several ways to access a given token. The choice of approach will depend on the token, its height, and the complexity of its family tree, which is prescribed in its descriptor. We give some examples of how a token might be accessed from its Work Space. We expect that you know ahead of time the token's family height, as this is an attribute of the port (Transition Input Port or Graph Output Port).

We assume that you already know how to access a token with height 0 or 1. Our first example shows how you might want to access a token with height 2. Without knowing specifically what you wish to do with the data in the token (construct an output token to be produced in a Transition Statement or display the data in a Command Program), we show code that prints out the data in the token along with its family indexing.

In these examples we assume that the variable `workspace` represents the Work Space of a Transition Input Port or Graph Output Port containing the token that we wish to access. We also assume that all index lower bounds are 0.

A.1 Code for access of a token with height 2:

```
/* WorkSpace methods */
unsigned int leafCount = workSpace.getNumLeaves();
unsigned int childCount = workSpace.getNumChildren();
unsigned int height = workSpace.getDepth();

/* output data */
cout << "leafCount " << leafCount << endl;
cout << "childCount " << childCount <<  endl;
cout << "height " << height << endl;
cout << endl;

/* Get the token from the workSpace and use Token Methods. */
/* These results should match the ones above from the WorkSpace. */
const GCL_Token_T<float> * myToken = workSpace.getToken();

unsigned int myLeafCount = myToken -> getLeafCount();
```

```
unsigned int myChildCount = myToken -> getChildCount();
unsigned int myHeight = myToken -> getDepth();

/* output data */
cout << "myLeafCount " << myLeafCount << endl;
cout << "myChildCount " << myChildCount << endl;
cout << "myHeight " << myHeight << endl;
cout << endl;

/* We know that the token has height 2, and we want to find out
   How many elements are in each row. */

/* Obtain the token's descriptor. */
const GCL_Descriptor * myDescriptor = myToken -> get__descriptor ();

/* Declare a vector to store the row sizes. */
vector<int> rowSize;
rowSize.resize (myChildCount);

/* Declare and initialize two index vectors: */
/* lowerIndexVector for the current child, */
/* upperIndexVector for the next child. */
/* We use index vectors with size 1 to specify only the first index.
   The descriptor function getOffset will find the give the offset to
   the beginning of the respective row.  */
vector<int> lowerIndexVector(1), upperIndexVector(1);

/* Get the row sizes from the descriptor. */
/* We can do this for all but the last row, which is a special case. */
for (int i = 0; i < myChildCount - 1; i++) {
  /* first index identifies the row */
  upperIndexVector[0] = i+1;
  lowerIndexVector[0] = i;
  rowSize[i] = myDescriptor -> getOffset(upperIndexVector)
    - myDescriptor -> getOffset(lowerIndexVector);
}

/* Get the last row size. */
rowSize[myChildCount - 1] =
  myLeafCount - myDescriptor -> getOffset(upperIndexVector);



/* Declare myIndexVector to access the leaves. */
vector<int> myIndexVector(2);

/* Print out the indices and leaf values of the token using a nested loop. */
cout << "token leaf values:" << endl;
for (int i = 0; i < myChildCount; i++) {
  if (rowSize[i] > 0) {
```

```
  cout << "  leaves in row " << i << ":" << endl;
  myIndexVector[0] = i;
  for (int j = 0; j < rowSize[i]; j++) {
    myIndexVector[1] = j;
    cout << "    leaf[" << i << "][" << j << "] = " <<
    myToken -> getLeaf(myIndexVector) <<
    endl;
  }
}
else cout << "  row " << i << " is empty" << endl;
```

A.2 Code for access of a token with height 3:

Accessing a token with height 3 is more involved, because the family tree at the lower levels is not immediately accessible directly from the token. Thus we "unpack" the token's descriptor to obtain a list (actually a *deque*, as defined in the STL) of the child descriptors, each of which will have height 2. We use these descriptors to uncover the family tree structure of the token and display its leaves.

```
/* WorkSpace methods */
unsigned int leafCount = workSpace.getNumLeaves();
unsigned int childCount = workSpace.getNumChildren();
unsigned int height = workSpace.getDepth();

/* output data */
cout << "leafCount " << leafCount << ", expecting 15." << endl;
cout << "childCount " << childCount << ", expecting 3." <<  endl;
cout << "height " << height << ", expecting 3." << endl;
cout << endl;

/* Get the token from the workSpace and use Token Methods. */
const GCL_Token_T<int> * myToken = workSpace.getToken();

unsigned int myLeafCount = myToken -> getLeafCount();
unsigned int myChildCount = myToken -> getChildCount();
unsigned int myHeight = myToken -> getDepth();
int myLowerBound = myToken -> getLowerBound();

/* output data */
cout << "myLeafCount " << myLeafCount << ", expecting 15." << endl;
cout << "myChildCount " << myChildCount << ", expecting 3." << endl;
cout << "myHeight " << myHeight << ", expecting 3." << endl;
cout << "myLowerBound " << myLowerBound << ", expecting 0." << endl;
cout << endl;

/* Here we prepare to output the stuff of the token. */

/* There are no functions defined that will give all of the
   index ranges directly from the highest level
   of a height 3 token.
   So we unpack the top level descriptor to get a deque of descriptors
   for the children, and we work from there. */

/* Note: We could simply unpack the tokens;
   However we give this approach as an alternative. */
const GCL_Descriptor * descriptor = workSpace.getDescriptor();

/* Get a deque of the descriptor's children. */
deque <const GCL_Descriptor *> * descriptorDeque
  = descriptor -> makeChildren();
```

```
/* NOTE: the descriptoreDeque is allocated within the above method,
   as are the descriptors in the deque.  To avoid a memory leak,
   we must remember to clear the deque and then delete it. */

/* declare an iterator for the descriptors in a deque */
deque <const GCL_Descriptor *>::iterator descriptorDequeIter;

/* Declare a vector to store the row sizes
   in each of the child descriptors. */
vector<int> rowSize;

/* Declare some other variables to be used for each descriptor in the deque. */
unsigned int myChildLeafCount;
unsigned int myChildChildCount;
unsigned int myChildHeight;

/* Declare two index vectors. */
/* lowerIndexVector for the current child. */
/* upperIndexVector for the next child. */
vector<int> lowerIndexVector(1), upperIndexVector(1);

/* Now go through each unpacked token in the same way as in the
   regular and irregular examples */

cout << "Data for token follows:" << endl;

/* Declare a descriptor pointer to be used for each descriptor in the deque. */
const GCL_Descriptor * myChildDescriptor;

/* Declare myIndexVector to access the leaves of the original token. */
vector<int> myIndexVector(3);

/* Loop through the descriptors in the deque */
int k = 0;
for (descriptorDequeIter = descriptorDeque -> begin();
     descriptorDequeIter != descriptorDeque -> end();
     descriptorDequeIter++, k++) {
  myChildDescriptor = *descriptorDequeIter;

  /* Get some data about the child. */
  myChildLeafCount = myChildDescriptor -> getLeafCount();
  myChildChildCount = myChildDescriptor -> getChildCount();
  myChildHeight = myChildDescriptor -> getDepth();

  /* We use index vectors with size 1 to specify only the first index.
     The descriptor function getOffset will give the offset in the data vector to
     the beginning of the respective row.  */

  if (myChildChildCount <= 0) cout << "Child [" << k << "] is empty." << endl;
```

```
  else {

    /* resize the rowSize vector */
    rowSize.resize (myChildChildCount);

    /* Get the row sizes from the descriptor. */
    /* We can do this for all but the last row, which is a special case. */
    for (int i = 0; i < myChildChildCount - 1; i++) {
      /* first index identifies the row */
      upperIndexVector[0] = i+1;
      lowerIndexVector[0] = i;
      rowSize[i] = myChildDescriptor -> getOffset(upperIndexVector)
        - myChildDescriptor -> getOffset(lowerIndexVector);
    }

    /* Now get the last row size. */
    rowSize[myChildChildCount - 1] =
      myChildLeafCount - myChildDescriptor -> getOffset(upperIndexVector);

    /* Print out the leaf values of the token in a 3-level nested loop. */
    myIndexVector[0] = k;
    cout << "leaves in child [" << k << "]:" << endl;
    for (int i = 0; i < myChildChildCount; i++) {
      if (rowSize[i] <= 0) cout << "descendant ["
                                << k << "][" << i << "] is empty." << endl;
      else {
        cout << "  leaves in [" << k << "][" << i << "]:" << endl;
        myIndexVector[1] = i;
        for (int j = 0; j < rowSize[i]; j++) {
          myIndexVector[2] = j;
          cout << "    leaf[" << k << "][" << i << "][" << j << "] = " <<
            myToken -> getLeaf(myIndexVector) <<
            endl;
        }
      }
    }
  }
}
clearDeque(*descriptorDeque);
delete descriptorDeque;
```

Appendix B
Examples: Construction of Tokens

In a Transition Statement, one will usually construct a token to be stored in a Work Space associated with a Transition Output Port. In a Command Program, the Work Space is associated with a Graph Input Port. The following examples assume that you are constructing a token to be contained in a Work Space.

There are several ways to construct a given token. The choice of approach will depend on the token, its height, and the complexity of its family tree, for which we construct its descriptor. We must construct a token whose family height matches that specified for the port (Transition Output Port or Graph Input Port).

We assume that you already know how to construct a token with height 0 or 1. Our first two examples show code to construct a token with height 2; the first example for a completely regular token, and the second for a token that is not regular. The third example shows code to construct a token with height 3: specifically the example cited in the PGM Spec.

In these examples we assume that the variable `workspace` represents the Work Space of a Transition Output Port or Graph Input Port to contain the token that we construct. We make sure that all index lower bounds are 0 (an exception to the example in the PGM Spec).

B.1 Code to construct a completely regular token with height 2

```
/************************************
  First we build the descriptor using a special constructor
  for regular families.
  Then we construct a token with that descriptor and
  populate its Data Vector before putting it in the workspace.

  Note that in the descriptor class, user-defined lower bounds
  for family indices are supported.
  The intent was for families of nodes, families
  of node ports, families of included graphs, etc.
  However for tokens we do not support user-defined lower bounds.
  In a token, all lower bounds for indices are set to 0.
  If the Command Program Writer or Transition writer constructs a token
  with a non-zero lower bound for an index and that token is unpacked
  via an unpack transition, the lower bound for the family index is lost.

  The unpack transition produces a sequence of tokens, each
  corresponding to a respective child of the input token.
  ************************************/
```

```
/* Declare a vector with two unsigned integers
   to specify the two dimensions */
vector<unsigned int> dimVector(2);

/* Then initialize it with the desired dimensions */
dimVector[0] = 2;  /* number of rows */
dimVector[1] = 3;  /* number of columns */

/* Create a regular descriptor */
/* Note, because this will be the descriptor of a token,
   we omit the second argument, allowing the default value of 0
   for the lower bound. */
GCL_Descriptor * descriptor = new GCL_Descriptor (dimVector);

/* Use this descriptor to create a token with base type float */
GCL_Token_T<float> * token = new GCL_Token_T<float> (descriptor);

/***********************************
   A WORD OF ADVICE:  The above allocates memory for a descriptor
   and returns a pointer to that descriptor.  This descriptor
   then becomes part of the token that is constructed.
   If you are in a transition statement and you want an output
   token to have a descriptor that is identical to the descriptor
   of an input token, you must make a copy of the input token's
   descriptor using the copy constructor.
   The reason is because the input token's descriptor will be
   automatically deleted after the Transition Statement has
   finished execution.
   ***********************************/

/* At this point we have a token whose data array is filled
   with float 0s.  So we now fill in the data. */

/* First, define an indexVector for accessing the data. */
vector<int> indexVector(2);

/* Then go through a nested loop to define the data */
for (int i = 0; i < 2; i++) {
  indexVector[0] = i;          /* set the row index */
  float baseValue = 39 + 10 * i;  /* set the base value */
  for (int j = 0; j < 3; j++) {
    indexVector[1] = j;        /* set the column index */
    /* set the leaf value and bump the baseValue for the next leaf */
    token -> putLeaf(indexVector, baseValue++);
  }
}

/* Finally, we put this token into the workSpace */
workSpace.storeToken(token);
```

B.2 Code to construct an irregular token with height 2

```
/************************************
Example to show how to build an irregular token of height 2
and store it in a WorkSpace.

We build a triangular token with four rows
(n-th row has n elements for row number n = 0, 1, 2, 3)
and store it in a workspace.
Note that the first row (with index 0) is empty.
************************************/

  /************************************
    We make the descriptor for each row and add it to
    a deque of descriptors.  This deque is then used to make the
    height 2 descriptor.
    ************************************/

  /* Declare a deque for the pointers to descriptors */
  deque<const GCL_Descriptor *> descriptorDeque;

  /* In a loop, construct a descriptor for each row and add it to the deque. */
  for (int i = 0; i < 4; i++) {
    /* This version of the GCL_Descriptor constructor constructs
       a descriptor for a height 1 family.
       The first argument (0) is the index lower bound.
       The second argument (i) is the number of leaves */
    descriptorDeque.push_back(new GCL_Descriptor(0,i));
  }

  /* Use the deque to create the height 2 descriptor. */
  /* The first argument is a deque of descriptors.
     The second argument is the common height of the descriptors
     in the deque. */
  GCL_Descriptor * descriptor = new GCL_Descriptor (descriptorDeque, 1);

  /* The deque is no longer needed, so clear it to prevent a memory leak. */
  clearDeque(descriptorDeque);

  /* The descriptorDeque was declared – not allocated - and thus will
     automatically be deleted when it goes out of scope. */

  /* Use this descriptor to create a token with base type float. */
  GCL_Token_T<float> * token = new GCL_Token_T<float> (descriptor);

  /************************************
     A WORD OF ADVICE:  The above allocates memory for a descriptor
     and returns a pointer to that descriptor.  This descriptor
     then becomes part of the token that is constructed.
```

```
   If you are in a transition statement and you want an output
   token to have a descriptor that is identical to the descriptor
   of an input token, you must copy the input token's descriptor.
   ************************************/

/* At this point we have a token whose data array is filled
   with float 0s.  So we now fill in the data. */

/* First, define an indexVector for accessing the data. */
vector<int> indexVector(2);

/* Then go through a nested loop to define the data */
for (int i = 0; i < 4; i++) {
  indexVector[0] = i;          /* set the row index */
  float baseValue = 39 + 10 * i;  /* set the base value */
  for (int j = 0; j < i; j++) {
    indexVector[1] = j;        /* set the column index */
    token -> putLeaf(indexVector, baseValue++);
       /* set the leaf value and bump for the next */
  }
}

/* Finally, we put this token into the workSpace */
workSpace.storeToken(token);
```

B.2 Code to construct an irregular token with height 3

This example shows code to construct a token that is almost the same as that shown in the PGM Spec on page 4. The major difference is that in this example, all index lower bounds are 0.

Reason: Because the Pack Transition does not support user selection of index lower bounds, and automatically sets all lower bounds to 0. Also, the Unpack Transition ignores the index lower bound when unpacking a token. The stream of tokens produced is produced to the respective output place in the order that they appear as children in the input token.

```
/************************************
Example to show how to build a token of height 3
and store it in a WorkSpace.

Except for the lower bound indices,
we build the example token shown on page 4 of the
PGM Specification, July 18, 2002.
In this example, we set all lower bound indices to 0.
************************************/

  /************************************
    We show how to build a token level by level.
    ************************************/

  /* The final token will have height 3, and we build it from height 2 tokens.
     So we build the height 2 tokens, each from height 1 tokens. */

  /* declare two deques -
     one for height 1 tokens and one for height 2 tokens. */
  deque <const GCL_Token_T<int> *> tokenDeque1;
  deque <const GCL_Token_T<int> *> tokenDeque2;

  /* declare an array for initial values
     this array size to accommodate the size of the largest height 1 token. */
  int dataValues[4];

  /* declare a pointer to a token with base type int. */
  GCL_Token_T<int> * token;

  /* declare an int for height one token size. */
  int tokenSize;

  /* Height 1 token[0][0] token has 3 children:  */
  tokenSize = 3;
  token = new GCL_Token_T<int>(new GCL_Descriptor(0,tokenSize));

  /* enter the initial values into the array. */
```

```
for (int i = 0; i < tokenSize; i++) {
  dataValues[i] = 39 + i;
}

/* initialize the token data. */
token -> setValues(dataValues, tokenSize);

/* store the token in the deque. */
tokenDeque1.push_back(token);

/* Height 1 token [0][1] also has 3 children:  */
tokenSize = 3;
token = new GCL_Token_T<int>(new GCL_Descriptor(0,tokenSize));

/* enter the initial values into the array. */
for (int i = 0; i < tokenSize; i++) {
  dataValues[i] = 49 + i;
}

/* initialize the token data. */
token -> setValues(dataValues, tokenSize);

/* store the token in the deque. */
tokenDeque1.push_back(token);

/* Now ready to create Height 2 [0] token
   from the deque of height 1 tokens.
   We pack the deque and store the resulting token
   in the deque of height 2 tokens. */
tokenDeque2.push_back ( pack (tokenDeque1, 1) );

/* Having done that, we clear the deque of height 1 tokens. */
/* Function deallocates the memory for the height 1 tokens
   and empties the deque. */
clearDeque (tokenDeque1);

/* Repeat the process for Height 2 [1] token
   Which happens to be empty.  */
tokenDeque2.push_back ( pack (tokenDeque1, 1) );

/* Don't really need to clear an empty deque (no problem if we did). */

/* Now for Height 2 [2] token.
   We repeat the above with fewer comments
   in a loop over the height 1 tokens */

/* Declare the base value */
int valueBase;

for (int i = 0; i < 4; i++) {
```

```
  switch (i) {
  case 0:
    tokenSize = 4;
    valueBase = 0;
    break;
  case 1:
    tokenSize = 3;
    valueBase = 9;
    break;
  case 2:
    tokenSize = 0;
    valueBase = 0;
    break;
  case 3:
    tokenSize = 2;
    valueBase = 28;
    break;
  }
  token = new GCL_Token_T<int>(new GCL_Descriptor(0,tokenSize));

  for (int j = 0; j < tokenSize; j++) {
    dataValues[j] = valueBase + j;
  }
  token -> setValues(dataValues, tokenSize);

  tokenDeque1.push_back(token);
}

/* Make Height 2 token [2] from the deque of tokens
   and add it to the deque. */
tokenDeque2.push_back ( pack (tokenDeque1, 1) );

/* Clear the deque. */
clearDeque (tokenDeque1);

/* Make the height 3 token and store it in the workspace. */
workSpace.storeToken( pack (tokenDeque2, 2));

/* Clear the deque of height 2 tokens. */
clearDeque (tokenDeque2);
```